A coinductive semantics of the Unlimited Register Machine

Alberto Ciaffaglione
Dipartimento di Matematica e Informatica
Università di Udine, Italia
alberto.ciaffaglione@uniud.it

We exploit (co)inductive specifications and proofs to approach the evaluation of low-level programs for the *Unlimited Register Machine (URM)* within the Coq system, a proof assistant based on the *Calculus of (Co)Inductive Constructions* type theory. Our formalization allows us to certify the implementation of partial functions, thus it can be regarded as a first step towards the development of a workbench for the formal analysis and verification of both converging and diverging computations.

1 Introduction

In this paper we report and discuss a formalization of the *Unlimited Register Machine* (URM) and its semantics within the *Calculus of (Co)Inductive Constructions* ($CC^{(Co)Ind}$).

The URM is a mathematical idealisation of a computer, one of the formal approaches to characterize the intuitive ideas of computability and decidability [12]. Programs for the URM are low-level, essentially assembly-like, and their execution gives rise to both converging and diverging computations. This is a typical situation where it is required to define and reason about *circular, potentially infinite* objects and concepts, *i.e.* systems with infinitely many states. Since structural induction trivially fails on these systems, one may resort to stronger approaches, such as, among other ones, *coinduction*.

Coinductive principles can be stated and exploited in different settings. From a *set-theoretical* standpoint coinduction arises when objects are viewed as *maximal fixed-points* of monotone operators, whereas the *categorical* approach is developed through *(final) coalgebras*. To develop the present work, we settle within the *logical* system of *Intuitionistic Type Theory*.

Actually, in intuitionistic type theory infinite objects are managed through *coinductive types*: these, roughly speaking, are collections of elements whose construction requires an infinite numbers of steps. In particular, a handy technique for dealing with coinductive definitions and proofs within $CC^{(Co)Ind}$ was introduced by Coquand [8] and refined by Giménez [17]. Although providing a limited form of coinduction, such an approach is particularly appealing, because *proofs* carried out by coinduction are accommodated as any other infinite, coinductively defined object. Remarkably, such a technique is mechanised in the system Coq [26]: this, one among the rare interactive environments that implement coinductive definition and proof principles, is an appreciated proof assistant, due to the fact that the automatization and the interaction with the user are well-balanced.

In this paper we formalize the URM and its semantics from the point of view of the *program certification*. In our opinion, such an encoding within a coinductive formal system, such as $CC^{(Co)Ind}$, has several benefits. First it is interesting *per se*, as experiments about the encoding of computability models are still lacking. Then it may be valuable in education, by giving the opportunity to undergraduate students (computability is actually a basic computer science course) to experiment with non-standard (*i.e.* coinductive) tools within a concrete, relatively simple application. Further it might be useful in the area of program transformations, because the formal treatment of low-level languages is mandatory to certify components of programming languages, such as type-checkers, interpreters, and compilers. Last but not

F. Yu and C. Wang: The 13th International Workshop on
Verification of Infinite State Systems (INFINITY'11)
EPTCS 73, 2011, pp. 49–63, doi:10.4204/EPTCS.73.7

© A. Ciaffaglione
This work is licensed under the
Creative Commons Attribution License.

the least, the present, novel theoretical case study witnesses the broad applicability of coinduction as a verification technique on infinite-state systems and the significance of its mechanisation.

Besides the points mentioned above, we claim that the originality of this paper relies also on the presentation of the encoding, which is illustrated and discussed without showing Coq code, but via the more abstract level of $CC^{(Co)Ind}$ (in any case, the Coq code is available to the interested reader at the web page of the author [7]), thus providing the reader with an extra pedagogic value.

In the next section we illustrate coinduction within $CC^{(Co)Ind}$; then in the following four sections we develop the formalization of the URM, dealing with programs, computations and functions; finally we discuss directions for further investigations in the light of what we achieve and of related work.

2 Coinduction in $CC^{(Co)Ind}$

The formal treatment of infinite objects and concepts is supported by $CC^{(Co)Ind}$ via the mechanism of coinductive types. These, by providing the user with a limited form of recursion, allow the formalization and the management of infinite data and infinite proofs.

First of all, one may define concrete, *infinite* objects (*i.e.* data) as elements of *coinductive types*, which are fully described by a set of *constructors*[1]. From a pure logical point of view, the constructors can be seen as *introduction rules*; these are interpreted coinductively, *i.e.* they are applied infinitely many times, hence the type being defined is inhabited by infinite objects:

$$\frac{s \in S}{0{:}s \in S}\,(0S)_\infty \qquad \frac{s \in S}{1{:}s \in S}\,(1S)_\infty$$

In this case we have formalized infinite sequences, *i.e.* *streams*, of bits, a coinductive type we name S. Optionally, coinductive types may contain finite objects too, that is, *potentially* infinite objects; in such a case also *constant* constructors, besides the recursive ones, have to be declared:

$$\frac{}{0 \in L}\,(0L) \qquad \frac{}{1 \in L}\,(1L) \qquad \frac{l \in L}{0{:}l \in L}\,(0L)_\infty \qquad \frac{l \in L}{1{:}l \in L}\,(1L)_\infty$$

So doing, we have defined L, the type of sequences of both finite and infinite length, *i.e.* *lazy lists*, of bits.

Once a new coinductive type is defined, the system provides automatically the *destructors*, *i.e.* an extension of the native pattern-matching capability, to *consume* the elements of the type itself. Therefore, coinductive types can also be viewed as the *largest* collection of objects closed *w.r.t.* the destructors.

Consistently with this intuition, the destructors *cannot* be used for defining functions by recursion on coinductive types, because their elements cannot be consumed down to a constant case. The natural way to allow self-reference is to consider the dual perspective of *building* individual, constant elements in coinductive types. Such a goal can be fullfilled through *lazy corecursive* functions:

$$\begin{array}{lcl}
\mathit{zeros} & \triangleq & 0{:}\mathit{zeros} \\
\mathit{odd}(s) & \triangleq & \text{match } s \text{ with } a{:}b{:}s' \Rightarrow a{:}\mathit{odd}(s') \\
\mathit{even}(s) & \triangleq & \text{match } s \text{ with } a{:}b{:}s' \Rightarrow b{:}\mathit{even}(s') \\
\mathit{merge}(s,t) & \triangleq & \text{match } s \text{ with } a{:}s' \Rightarrow \text{match } t \text{ with } b{:}t' \Rightarrow a{:}b{:}\mathit{merge}(s',t')
\end{array}$$

Corecursive functions produce infinite objects and may have any type as domain (note that in the last three definitions we have applied the *match* destruction operation on a parameter of the domain). Infinite

[1] The constructors must respect a *strict positivity constraint* condition to guarantee the reduction termination of the calculus.

objects are not unfolded, unless their components are explicitly needed, "on demand", by a destruction operation. Therefore, to prevent the evaluation of corecursive functions from infinitely looping, their definition must satisfy a *guardedness condition*: every corecursive call has to be guarded by at least one constructor, and by nothing but constructors[2]. This way of regulating the implementation of corecursion captures the intuition that infinite objects are built via the iteration of an initial step.

Given a concrete coinductive type (such as S and L above), no proof principle can be automatically generated by the system: in fact, proving properties about infinite objects requires the potential of building *proofs* which are infinite as well! What is needed is the design of *ad-hoc* coinductive *predicates*, *i.e.* coinductive *propositions*, which are actually inhabited by such *infinite proofs*[3]. The traditional example is point-wise equality (also known as *bisimilarity*), that we define on streams and name $\simeq \subseteq S \times S$:

$$\frac{b{\in}\{0,1\} \quad s \simeq t}{b{:}s \simeq b{:}t}\,(\simeq)_\infty$$

Two streams are bisimilar if we can *observe* that they have equal heads and recursively, *i.e.* *coinductively*, their tails are bisimilar. Once this new predicate is defined, the system provides the corresponding *proof principle*, to carry out proofs about bisimilarity: such a tool, named *guarded induction* principle [8, 17], is particularly appealing in a context where proofs are managed as any other infinite object.

In fact, a proof by guarded induction is just an infinite object built by lazy corecursion (hence it must respect the same guardedness constraint that lazy corecursive functions have to). Remarkably, the mechanization of the guarded induction principle provides a handy technique for the construction of infinite proofs, which can be carried out interactively through the `cofix` tactic[4]. This tactic allows to build infinite proofs as *infinitely regressive* proofs, by assuming the thesis as an extra hypothesis and using it carefully later, provided its application is guarded by constructors. This "internal" approach is very direct, compared to the traditional techniques based on bisimulations, because the proofs do not need to be exhibited beforehand, but can be built incrementally via tactics.

To illustrate the support provided by the `cofix` tactic, we pick out the following coinductive property:

$$\forall s{\in}S.\ \mathit{merge}(\mathit{odd}(s),\ \mathit{even}(s)) \simeq s$$

We prove this proposition by mimicking the top-down proof practice of CC$^{\text{(Co)Ind}}$. First, the coinductive hypothesis is assumed among the hypotheses and the stream s is destructed two times into $a{:}b{:}t$; then the corecursive functions *odd*, *even* and *merge*, in turn, may perform a computation step; finally the constructor $(\simeq)_\infty$ is applied twice. In the end, we have reduced the goal to prove $\mathit{merge}(\mathit{odd}(t),\ \mathit{even}(t)) \simeq t$, a proposition which is an instance of the coinductive hypothesis. Therefore one is eventually allowed to exploit the coinductive hypothesis itself, whose application is now guarded by the constructor $(\simeq)_\infty$. The application of the coinductive hypothesis completes the proof, and intuitively has the effect of repeating ad infinitum the explicit, initial proof segment, thus realizing the "and so on forever" motto.

To avoid ambiguity with genuine induction, we say that the proof has been performed by *structural*

[2] Syntactically, the constructors guard the recursive call "on the left".

[3] This distinction between concrete objects and proofs points out that sets inhabited by concrete objects have *computational* content, whereas predicates inhabited by proofs carry *logical* information.

[4] A tactic is a command to solve a goal or decompose it into simpler goals.

coinduction on the derivation. The whole proof may be displayed in natural deduction style[5] as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[merge(odd(t),\ even(t)) \simeq t]_{(1)}}{a{:}b{:}merge(odd(t),\ even(t)) \simeq a{:}b{:}t}(\simeq)_\infty}{merge(a{:}odd(t),\ b{:}even(t)) \simeq a{:}b{:}t}(computation{:}\ merge)}{merge(odd(a{:}b{:}t),\ even(a{:}b{:}t)) \simeq a{:}b{:}t}(computation{:}\ odd,\ even)}{merge(odd(s),\ even(s)) \simeq s}(destruction)}{\forall s{\in}S.\ merge(odd(s),\ even(s)) \simeq s}(introduction)}{\forall s{\in}S.\ merge(odd(s),\ even(s)) \simeq s}(1)$$

To conclude, we observe that, as the reader may imagine, there exist several *semantically* productive[6], but *syntactically* non-guarded functions (and proofs) that cannot be accepted by $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$, because the automated check is not sophisticated enough. Particular effort is put in fact by the community into the goal of extending the expressive power of guarded corecursion [19, 16, 5]. At the moment, we can say that $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$ has made a lot of progress, but there are still problematic issues on the carpet.

3 The Unlimited Register Machine

The Unlimited Register Machine (URM) is a mathematical idealisation of a computer, one among the frameworks proposed to set up a formal characterisation of the intuitive ideas of effective computability and decidability. It is equivalent to the alternative approaches, *e.g.* Turing machines, and particulary valued for its simplicity. We work here with the URM formulation introduced by Cutland [12], a slight variation of a machine first conceived by Shepherdson and Sturgis [23].

Registers and instructions. The URM has an *infinite* number of *registers* $R_1, R_2, \ldots$ containing natural numbers $r_1, r_2, \ldots$ which may be altered by *instructions*. These are of four kinds and have the following intended meaning ($r \to R$ represents the loading of the natural value r in the register R):

$Z(i)$	$\triangleq$	Zero	:	$0 \to R_i$
$S(i)$	$\triangleq$	Successor	:	$r_i + 1 \to R_i$
$T(i,j)$	$\triangleq$	Transfer	:	$r_i \to R_j$
$J(i,j,k)$	$\triangleq$	Jump	:	if $r_i{=}r_j$ then proceed from the kth instruction else proceed from the next instruction

Programs and computations. A *program* for the URM is a finite, non-empty sequence of instructions.

When provided with a program P and a(n *initial) configuration* (*i.e.* a *finite*, non-empty sequence of natural numbers $r_1, r_2, \ldots, r_m$ in the registers $R_1, R_2, \ldots, R_m$)[7], the URM performs a *computation*: this means starting from the first instruction in P and obeying the instructions sequencially (unless a Jump is encountered), thus altering at any step the content of the registers as prescribed by the instructions.

[5] As usual, local hypotheses are indexed with the rules they are discharged by.

[6] Productivity is the power of a function call to produce data, which is undecidable.

[7] Despite the number of the registers being infinite, any program P is finite, so there exists a maximal register index $m{=}\rho(P)$, depending on P, such that R_m is affected by the instructions in P. Hence $r_1, r_2, \ldots, r_m$ is equivalent to $r_1, r_2, \ldots, r_m, 0, 0, \ldots$

The computation *stops*, or *converges*, if and only if there is no next instruction; when this is the case, the number r stored in R_1 in the *final* configuration is regarded as the output of the computation, and this is written $P(r_1, r_2, \ldots, r_m)\downarrow r$. On the other hand, due to the looping back via the Jump instruction, there are computations that *never stop*, or *diverge*, which is written $P(r_1, r_2, \ldots, r_m)\uparrow$.

Formalization in CC$^{\text{(Co)Ind}}$. The encoding of the basic URM structures in CC$^{\text{(Co)Ind}}$ is straightforward, because both configurations and programs are simply finite, non-empty sequences of components, which we formalize by means of inductive datatypes (the $\mathbb{N}$ represents the natural numbers):

Loc	:	i, j	$\in$	$\mathbb{N}^+ = \mathbb{N} - \{0\}$	register index
Val	:	r	$\in$	$\mathbb{N}$	register content
Cgn	:	σ	$::=$	$(\iota \mapsto r_\iota)^{\iota \in [1..m]}$	list-configuration
PC	:	k, h	$\in$	$\mathbb{N}$	program counter
$Inst$	:	I	$\in$	$\{Z(i),\ S(i),\ T(i,j),\ J(i,j,k)\}$	instruction
Pgm	:	U, V	$::=$	$\langle \iota \mapsto I_\iota \rangle^{\iota \in [1..n]}$	program

An alternative encoding of configurations can be given via *infinite* sequences, *i.e.* coinductive datatypes:

$$Cgn_\infty \ : \ \sigma_\infty \ ::= \ (\iota \mapsto r_\iota)^{\iota \in [1..\infty]} \qquad \text{stream-configuration}$$

Adequacy (I). We start to address now the faithfulness of our encoding of the URM, by comparing Cutland's formulation and our formalization in CC$^{\text{(Co)Ind}}$. First, we observe that the syntax of our instructions (and therefore of programs) coincide with Cutland's one. Then, two technical points have to be considered: about the convergence of computations, and about the encoding of configurations.

The "natural" way for the program $U = I_1, I_2, \ldots, I_n$ to stop is that the program counter is set eventually to $n+1$; though, a Jump instruction could set it to an index greater than $n+1$. Cutland actually confines his attention to the programs that invariably stop because the next instruction should be I_{n+1}. We adopt a similar convention here, with the difference that we use the index 0 in place of $n+1$: these kinds of programs, the sole we will be considering from now on, are said to be *in standard form*.

Definition 3.1 *(Standard form)*
A program $U = \langle \iota \mapsto I_\iota \rangle^{\iota \in [1..n]}$ *is* in standard form *if, for every* $J(i, j, k) \in U$, $k \leq n$ *holds.*

As far as the formalization of configurations is concerned, it is apparent that our stream-configurations (*i.e.* the datatype Cgn_∞) correspond to infinite sequences of registers in the original URM.

By working *on paper*, on the one hand, Cutland is naturally allowed to define configurations as finite, starting segments of such infinite sequences of registers: in fact, by inspecting a given program P, one can pick out $\rho(P)$, the maximal register index affected by the instructions in P. In this way the working space available to the computation under P may be restricted to the configuration $r_1, r_2, \ldots, r_{\rho(P)}$.

On the other hand, working *formally* within CC$^{\text{(Co)Ind}}$ requires extra care. First we observe that our list-configurations (*i.e.* the datatype Cgn) correspond to the above Cutland configurations $r_1, r_2, \ldots, r_{\rho(P)}$. Nevertheless, list-configurations bring a drawback: if one wants to reason formally on them, it is required to consider only programs that respect the working space they make available[8]. That is, programs and list-configurations can be soundly coupled just if the programs contain "good" pointers (*i.e.* indexes) to the configurations themselves, a constraint that can be viewed as a kind of *compatibility* concept.

[8] In a sense, this means to provide in advance with the maximal register index $\rho(U)$, given a program U.

Definition 3.2 *(Compatibility) A program U and a list-configuration $\sigma=(\iota\mapsto r_\iota)^{i\in[1..m]}$ are* compatible *($\sigma \models U$) if U is in standard form and, for every $Z(i)$, $S(i)$, $T(i,j)$, $J(i,j,k)\in U$, $i,j\in[1..m]$ holds.*

4 Abstract computation

In this section we bootstrap the semantics of the URM, by extending in a modular way the formalization introduced so far; note that, from now on, we will use the terminology "configuration" to refer to the encoding in $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$ itself (*i.e.* either the finite-list datatype *Cgn* or the infinite-stream datatype Cgn_∞).

It is apparent that the concept of *convergence* of computations can be relativised *w.r.t.* configurations: there are actually programs that always stop and programs that never stop (whatever configuration is coupled to them) and programs that either converge or diverge depending on the initial configuration. Clearly, the divergence is caused by the presence of *infinite loops* in the progress of computation: to deal formally with the execution of programs we have then to manage an infinite-state system, a scenario which may benefit from the use of the *coinduction* as a specification and proof principle.

In this section we focus just on a restricted, basic notion of computation: in fact, from the point of view of the termination, the only essential instruction is the *Jump* instruction, which has the capability to *separate* converging computations from diverging ones. Hence we consider here programs that contain only Jump instructions, *i.e. abstract* programs; this preliminary investigation allows us to focus on the object system from a cleaner perspective, to be exploited in the following.

Noticeably, it is not possible to cope with the semantics of URM programs by using a unique, *potentially* coinductive computation concept (see Section 2): a faithful encoding has actually to reflect the separation between converging and diverging computations, through two different judgments. Therefore, using in this case *finite* (*i.e.* list) configurations, the semantics of abstract URM programs can be described by the *inductive* cp_{j+} and the *coinductive* $cp_{j\infty}$ predicates, whose arity is $Pgm\times Cgn\times PC$.

Definition 4.1 *(Abstract evaluation) Let $A=\langle\iota\mapsto I_\iota\rangle^{\iota\in[1..n]}$ and $\sigma=(\iota\mapsto r_\iota)^{\iota\in[1..m]}$ be an abstract program and a configuration such that $\sigma\models A$, and let $h\in[1..n]$ and $I_h=J(i,j,k)$. Then, cp_{j+} is defined by the first four rules, interpreted inductively, and $cp_{j\infty}$ by the last two rules, interpreted coinductively:*

$$\frac{h=n \quad r_i\neq r_j}{cp_{j+}(A,\sigma,h)}\,(f\cdot l)_+ \qquad \frac{k=0 \quad r_i=r_j}{cp_{j+}(A,\sigma,h)}\,(t\cdot l)_+$$

$$\frac{cp_{j+}(A,\sigma,h+1) \quad h<n \quad r_i\neq r_j}{cp_{j+}(A,\sigma,h)}\,(f\cdot r)_+ \qquad \frac{cp_{j+}(A,\sigma,k) \quad k\neq 0 \quad r_i=r_j}{cp_{j+}(A,\sigma,h)}\,(t\cdot r)_+$$

$$\frac{cp_{j\infty}(A,\sigma,h+1) \quad h<n \quad r_i\neq r_j}{cp_{j\infty}(A,\sigma,h)}\,(f\cdot r)_\infty \qquad \frac{cp_{j\infty}(A,\sigma,k) \quad k\neq 0 \quad r_i=r_j}{cp_{j\infty}(A,\sigma,h)}\,(t\cdot r)_\infty$$

At the moment, our goal is to capture just the progress of the *control flow*, with the computation that may proceed from a generic instruction of a program. Specifically, the intended meaning of the judgments $cp_{j+}(A,\sigma,h)$ and $cp_{j\infty}(A,\sigma,h)$ is that the computation under the abstract program A with the configuration σ and by starting from the hth instruction of A, *converges* and *diverges*, respectively.

More in detail, the coinductive predicate asserts that the computation loops: that is, by starting from the instruction I_h, there exists an instruction I_q which can be reached from I_h and such that, afterwards,

the control flow comes again at I_q after a non-zero, finite number of steps. Hence, the divergence is grasped via the predicate cp_∞ by the coinduction proof principle motto ("and so on forever").

We remark that, since URM programs are not structured, we have to embed in the encoding some other "structuration" criterium; in fact, the design of the predicates has been directly inspired by the *number of evaluation steps* implicit amount. Thus we have defined two atomic rules for cp_{j+} (the evaluation stops in one step), when either the current one is the last instruction and the Jump condition is false, or the current Jump condition is true and the instruction tells to jump out of the program. The extra rules are recursive, and address how an evaluation step is carried out within a converging computation (predicate cp_{j+}) and a diverging one (predicate $cp_{j\infty}$), again inspecting by cases the Jump condition.

Another important choice to be pointed out is that we have modeled the evaluation from a particular perspective, *i.e.* for using the judgments, according to Coq's top-down proof practice, to *execute* specific programs. This "algorithmic" approach is motivated by the fact that we are interested in experimenting the certification of concrete programs; this is a preliminary step that pinpoints further investigations, such as the development of the metatheory of the URM or the advanced issues addressed by Leroy and Grall [22]. We are conscious that these more ambitious tasks could require the introduction of new versions of the evaluation concept, to be related to the ones we have formalized up to date.

We notice, finally, that a fragment of the encoding of the evaluation judgments, which is common to all the rules, has not been displayed in the rules themselves, but has been collected within the hypotheses of the Definition 4.1: such a part of the formalization has to cope with the compatibility between programs and finite configurations, an overhead that we have discussed in the previous section.

In the end, using our machinery we can manage termination and divergence of computations under abstract URM programs parameterically *w.r.t.* non-mutable configurations, as follows.

Definition 4.2 *(Converging and diverging abstract evaluation) Let A and σ be an abstract program and a configuration such that $\sigma \models A$. The computation under A with σ* converges *and* diverges *when:*

$$\begin{array}{lcl} stop_j(A,\sigma) & \triangleq & cp_{j+}(A,\sigma,1) \\ loop_j(A,\sigma) & \triangleq & cp_{j\infty}(A,\sigma,1) \end{array}$$

As an example, let us consider the abstract program $B \triangleq \langle 1 \mapsto J(1,2,2),\ 2 \mapsto J(1,2,2)\rangle$. We can prove that the computation under B with the configuration $\sigma \triangleq (1 \mapsto 0,\ 2 \mapsto 1)$ converges, while it diverges with $\tau \triangleq (1 \mapsto 0,\ 2 \mapsto 0)$; both the proofs are immediate, the second one is by coinduction[9]:

$$\dfrac{r_1{=}0{\neq}1{=}r_2 \quad \dfrac{r_1{=}0{\neq}1{=}r_2}{cp_{j+}(B,\sigma,2)}\,(f{\cdot}l)_+}{cp_{j+}(B,\sigma,1)}\,(f{\cdot}r)_+ \qquad \dfrac{2{\neq}0 \quad r_1{=}0{=}r_2 \quad \dfrac{2{\neq}0 \quad r_1{=}0{=}r_2 \quad [cp_{j\infty}(B,\tau,2)]_{(1)}}{cp_{j\infty}(B,\tau,2)}\,(t{\cdot}r)_\infty(1)}{cp_{j\infty}(B,\tau,1)}\,(t{\cdot}r)_\infty$$

A more sensible approach would allow to manage *variable* configurations, such as $\mu \triangleq (1 \mapsto m,\ 2 \mapsto n)$. In that case, the Definition 4.2 should be more involved, by including a premise to *constrain* the content of the configuration at hand. So doing, one could prove more general assertions, such as *e.g.* $(m{\neq}n) \Rightarrow cp_{j+}(B,\mu,1)$ and $(m{=}n) \Rightarrow cp_{j\infty}(B,\mu,1)$. Though, we prefer to postpone such versions of convergence and divergence to the next section, where we will address the full URM instruction suite.

[9] As discussed in Section 2, the proofs are displayed in natural deduction style and have to be read from the bottom.

5 Full computation

We extend now our formalism to deal with the full URM, by adopting *infinite* (*i.e.* stream) configurations, because these allow to dispose of the compatibility between programs and configurations themselves (as argued in Sections 3 and 4). Note that the results we get are independent from the particular encoding of configurations (in fact, at the end of this section we will relate formally finite and infinite configurations to each other, by addressing the adequacy of the whole formalization).

Actually, the computation under URM programs is captured by the more involved inductive predicate cp_+, with arity $Pgm \times Cgn_\infty \times PC \times Cgn_\infty$, and the coinductive predicate cp_∞, with arity $Pgm \times Cgn_\infty \times PC$, which describe *both* the control flow *and* its effect on configurations.

Definition 5.1 *(Evaluation) Let* $U=\langle \iota \mapsto I_\iota \rangle^{\iota \in [1..n]}$ *and* $\sigma_\infty=(\iota \mapsto r_\iota)^{\iota \in [1..\infty]}$ *be a program and a configuration, and let* $h \in [1..n]$*. We assume that* $I_h=J(i,j,k)$ *in the Jump rules (those labelled* $(j-)$*),* $I_h=Z(i)$ *in the Zero rules,* $I_h=S(i)$ *in the Successor rules, and* $I_h=T(i,j)$ *in the Transfer rules.*

Then, cp_+ *is defined by the following rules, interpreted inductively:*

$$\frac{h{=}n \quad r_i{\neq}r_j}{cp_+(U,\sigma_\infty,h,\sigma_\infty)}\,(jf{\cdot}l)_+ \qquad \frac{cp_+(U,\sigma_\infty,h{+}1,\tau_\infty) \quad h{<}n \quad r_i{\neq}r_j}{cp_+(U,\sigma_\infty,h,\tau_\infty)}\,(jf{\cdot}r)_+$$

$$\frac{k{=}0 \quad r_i{=}r_j}{cp_+(U,\sigma_\infty,h,\sigma_\infty)}\,(jt{\cdot}l)_+ \qquad \frac{cp_+(U,\sigma_\infty,k,\tau_\infty) \quad k{\neq}0 \quad r_i{=}r_j}{cp_+(U,\sigma_\infty,h,\tau_\infty)}\,(jt{\cdot}r)_+$$

$$\frac{h{=}n \quad \tau_\infty{=}zr(\sigma_\infty,i)}{cp_+(U,\sigma_\infty,h,\tau_\infty)}\,(z{\cdot}l)_+ \qquad \frac{cp_+(U,\sigma'_\infty,h{+}1,\tau_\infty) \quad h{<}n \quad \sigma'_\infty{=}zr(\sigma_\infty,i)}{cp_+(U,\sigma_\infty,h,\tau_\infty)}\,(z{\cdot}r)_+$$

$$\frac{h{=}n \quad \tau_\infty{=}sc(\sigma_\infty,i)}{cp_+(U,\sigma_\infty,h,\tau_\infty)}\,(s{\cdot}l)_+ \qquad \frac{cp_+(U,\sigma'_\infty,h{+}1,\tau_\infty) \quad h{<}n \quad \sigma'_\infty{=}sc(\sigma_\infty,i)}{cp_+(U,\sigma_\infty,h,\tau_\infty)}\,(s{\cdot}r)_+$$

$$\frac{h{=}n \quad \tau_\infty{=}mv(\sigma_\infty,i,j)}{cp_+(U,\sigma_\infty,h,\tau_\infty)}\,(t{\cdot}l)_+ \qquad \frac{cp_+(U,\sigma'_\infty,h{+}1,\tau_\infty) \quad h{<}n \quad \sigma'_\infty{=}mv(\sigma_\infty,i,j)}{cp_+(U,\sigma_\infty,h,\tau_\infty)}\,(t{\cdot}r)_+$$

And cp_∞ *is defined by the following rules (a superset of those for* $cp_{j\infty}$*), interpreted coinductively:*

$$\frac{cp_\infty(U,\sigma_\infty,h{+}1) \quad h{<}n \quad r_i{\neq}r_j}{cp_\infty(U,\sigma_\infty,h)}\,(jf{\cdot}r)_\infty \qquad \frac{cp_\infty(U,\sigma_\infty,k) \quad k{\neq}0 \quad r_i{=}r_j}{cp_\infty(U,\sigma_\infty,h)}\,(jt{\cdot}r)_\infty$$

$$\frac{cp_\infty(U,\tau_\infty,h{+}1) \quad h{<}n \quad \tau_\infty{=}zr(\sigma_\infty,i)}{cp_\infty(U,\sigma_\infty,h)}\,(z{\cdot}r)_\infty \qquad \frac{cp_\infty(U,\tau_\infty,h{+}1) \quad h{<}n \quad \tau_\infty{=}sc(\sigma_\infty,i)}{cp_\infty(U,\sigma_\infty,h)}\,(s{\cdot}r)_\infty$$

$$\frac{cp_\infty(U,\tau_\infty,h{+}1) \quad h{<}n \quad \tau_\infty{=}mv(\sigma_\infty,i,j)}{cp_\infty(U,\sigma_\infty,h)}\,(t{\cdot}r)_\infty$$

The corecursive[10] *functions* $zr,sc,mv : Cgn_\infty \times \mathbb{N}^+(\times\mathbb{N}^+) \to Cgn_\infty$ *alter the configurations, as pre-*

[10] Corecursion is defined in Section 2. Note that these functions would be *recursive* working with finite configurations.

scribed by the instructions Zero, Successor and Transfer; the definition of zr is e.g. as follows[11]*:*

$$zr(\sigma_\infty, i) \triangleq \textit{match } \sigma_\infty \textit{ with } r : \tau_\infty \Rightarrow \textit{match } i{-}1 \textit{ with } 0 \Rightarrow 0 : \tau_\infty \mid n{+}1 \Rightarrow r : zr(\tau_\infty, i{-}1)$$

The intended meaning of the judgment $cp_+(U, \sigma_\infty, h, \tau_\infty)$ is that the computation under the program U with the configuration σ_∞ and by starting from the hth istruction of U, *stops*, transforming σ_∞ into τ_∞.

On the other hand, the intended meaning of $cp_\infty(U, \sigma_\infty, h)$ is the same as cp_{j_∞}, even if the configurations may be updated, in the case: the computation under the program U with the configuration σ_∞ and by starting from the hth istruction, *loops*. That is, there exists an instruction I_q which can be reached from I_h and such that, afterwards, the control flow comes again at I_q after a non-zero, finite number of steps. Nevertheless, the use of cp_∞ is subtler than that of cp_{j_∞}: the coinductive hypothesis ("and so on forever") may be actually applied, to grasp the divergence, *provided* the configuration at hand satisfies an *invariant* (whose nature will be clarified below). Coherently with such an intuition, a *final* configuration (corresponding to the fourth parameter of the inductive predicate cp_+) *cannot* exist for cp_∞, simply because the configurations may be updated "ad infinitum" in the course of a diverging computation!

Termination and divergence are now fully significant, and managed parameterically as follows.

Definition 5.2 *(Converging and diverging evaluation) Let U and σ_∞ be a program and a configuration, and let $\mathscr{T}(\sigma_\infty, U)$, $\mathscr{I}(\sigma_\infty, U)$ be decidable constraints about the content of the registers in σ_∞, depending on U. Then, the computation under U with σ_∞* converges *and* diverges *when, respectively:*

$$\begin{array}{lcl} stop(U, \sigma_\infty) & \triangleq & \exists \tau_\infty,\ \exists \mathscr{T}(\sigma_\infty, U).\ \mathscr{T}(\sigma_\infty, U) \Rightarrow cp_+(U, \sigma_\infty, 1, \tau_\infty) \\ loop(U, \sigma_\infty) & \triangleq & \exists \mathscr{I}(\sigma_\infty, U).\ \mathscr{I}(\sigma_\infty, U) \Rightarrow cp_\infty(U, \sigma_\infty, 1) \end{array}$$

As foreseen by the above comments about cp_+ and cp_∞, the management of convergence and divergence are fairly different between each other, when the configurations can be updated by computations.

Converging computations under U with initial σ_∞ are actually accommodated in the intuitive way: the halting is described by the program counter, which is eventually set to 0; moreover, the incremental modification of σ_∞ is reported in the final τ_∞. The premise $\mathscr{T}(\sigma_\infty, U)$ plays the role of a *termination* condition, which, if needed, provides with the extra potential of carrying out proofs by induction. In fact, computations may converge essentially in two ways: with or without the presence of *finite cycles*. In the latter case, the constraint just "guides" the control flow to the end of the program; in the presence of cycles, it is exploited to pick out a parameter on which to reason by induction. Therefore, in our *logical* setting, program-driven termination constraints make feasible formal proofs about the convergence and the output of individual programs *w.r.t.* parameter configurations. In other words, such conditions allow to make formal the informal proofs by evidence that one may figure out by inspecting the programs.

Conversely, the modification of the starting configuration σ_∞ within diverging computations under U does not produce a final configuration, because σ_∞ is updated ad infinitum. Though, the modification of σ_∞ *can be observed* in the course of the computation, and such configuration may be checked against an *invariance* condition, that constrains its content. Therefore, the invariance condition $\mathscr{I}(\sigma_\infty, U)$ itself, whose shape depends again on U, becomes the "guard" to ensure the non-termination[12].

Concerning the termination and invariance constraints, we restrict to universally quantified formulas on natural numbers, built via the logical operators and the arithmetic operations and predicates.

[11] We use here the notation $r : \sigma_\infty$ to represent the configuration $(0 \mapsto r,\ \imath \mapsto r_\imath)^{\imath \in [2..\infty]}$.

[12] We remark that the whole scenario is coherent *w.r.t.* the concept of *computable function*, that we will address in Section 6: there is an output, which is extracted from the final τ_∞, if and only if a computation stops.

For the sake of illustrating the technical details, let us consider the *parametric* (*i.e.* variable-content) configuration $\mu_\infty \triangleq (1\mapsto m,\ 2\mapsto n,\ 3\mapsto p,\ldots)$ and the program $V \triangleq \langle 1\mapsto S(1),\ 2\mapsto J(2,3,1)\rangle$. We can then show that the computation under V with μ_∞ diverges, by choosing the invariant $n{=}p$ (while it converges with the termination constraint $n{\neq}p$). To prove $\forall m,n,p.\ (n{=}p) \Rightarrow cp_\infty(V,\mu_\infty,1)$ by structural coinduction on the derivation within Coq's top-down proof environment, we assume in the proof context the coinductive hypothesis, the variables and the invariant; then we execute the two instructions of U so that the control flow loops back to the first instruction; finally we apply the coinductive hypothesis[13], which demands to prove that the new configuration satisfies the invariant constraint as well:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[n{=}p]}{\vdots}}{n{=}p}}{[cp_\infty(U,(1\mapsto m{+}1,\ 2\mapsto n,\ 3\mapsto p,\ldots),1)]_{(1)}}}{cp_\infty(U,(1\mapsto m{+}1,\ 2\mapsto n,\ 3\mapsto p,\ldots),2)}(jt{\cdot}r)_\infty}{cp_\infty(U,(1\mapsto m,\ 2\mapsto n,\ 3\mapsto p,\ldots),1)}(s{\cdot}r)_\infty}{\forall m,n,p{\in}\mathbb{N}.\ (n{=}p) \Rightarrow cp_\infty(U,(1\mapsto m,\ 2\mapsto n,\ 3\mapsto p,\ldots),1)}(introduction)$$
$$\dfrac{\forall m,n,p{\in}\mathbb{N}.\ (n{=}p) \Rightarrow cp_\infty(U,(1\mapsto m,\ 2\mapsto n,\ 3\mapsto p,\ldots),1)}{\forall m,n,p{\in}\mathbb{N}.\ (n{=}p) \Rightarrow cp_\infty(U,(1\mapsto m,\ 2\mapsto n,\ 3\mapsto p,\ldots),1)}(1)$$

Adequacy (II). We complete now the discussion about the faithfulness of our encoding *w.r.t.* Cutland's URM [12], undertaken in Section 3: the issues we have to address formally are the relationship between finite and infinite configurations, and the semantics given in the current and the previous section.

As far as the configurations are concerned, we first define the *inclusion* and *restriction* concepts.

Definition 5.3 *(Configuration inclusion/restriction) Let U be a program, $\sigma{=}(\iota\mapsto s_\iota)^{\iota\in[1..m]}$ a finite configuration and $\tau_\infty{=}(\iota\mapsto t_\iota)^{\iota\in[1..\infty]}$ an infinite one. Then,* inclusion *and* restriction *are defined as follows:*

$$\begin{array}{lcl} \sigma \subset \tau_\infty & \triangleq & (\forall \iota{\in}[1..m].\ t_\iota{=}s_\iota) \wedge (\forall \iota{>}m.\ t_\iota{=}0) \\ \tau_{\infty|U} & \triangleq & (\iota\mapsto t_\iota)^{\iota\in[1..\rho(U)]} \end{array}$$

Concerning the semantics, let us assume (without displaying the rules) to have introduced a second definition fot both the predicates cp_+ and cp_∞, to cope with *finite* configurations and for which we use an overloaded notation. The new rules differ from Definition 5.1 only for the fact that the involved finite configurations require the extra compatibility constraint with programs, analogously to Definition 4.1.

Now we can state the *equivalence* between finite and infinite configurations encodings *Cgn* and Cgn_∞.

Theorem 5.4 *(Configurations equivalence) Let $U{=}\langle\iota\mapsto I_\iota\rangle^{\iota\in[1..n]}$ be a program, σ and τ finite configurations, σ_∞ and τ_∞ infinite configurations, and let $h{\in}[1..n]$. Then the following properties hold:*

1. $cp_+(U,\sigma,h,\tau) \wedge \sigma{\models}U \wedge \sigma{\subset}\sigma_\infty \wedge \tau{\subset}\tau_\infty \Rightarrow cp_+(U,\sigma_\infty,h,\tau_\infty)$
2. $cp_\infty(U,\sigma,h) \wedge \sigma{\models}U \wedge \sigma{\subset}\sigma_\infty \Rightarrow cp_\infty(U,\sigma_\infty,h)$
3. $cp_+(U,\sigma_\infty,h,\tau_\infty) \Rightarrow cp_+(U,\sigma_{\infty|U},h,\tau_{\infty|U})$
4. $cp_\infty(U,\sigma_\infty,h) \Rightarrow cp_\infty(U,\sigma_{\infty|U},h)$

[13] The application of the coinductive hypothesis is *guarded* by the two constructors $(s{\cdot}r)_\infty$ and $(jt{\cdot}r)_\infty$ (see also Section 2).

PROOF. *(1, 3) By induction on the evaluation hypothesis. (2, 4) By coinduction on the derivation.*

Even if the above Theorem establishes that working either with finite, list-like configurations or with infinite, stream-like ones, is equivalent, we have preferred up to date to handle *infinite* configurations. Our choice is motivated by two reasons: stream configurations do not require the overhead of managing side-conditions to model the compatibility with programs, and it has not been yet necessary to perform proofs by induction on the structure of configurations themselves.

In the end, the reader can see that our machinery provides the user with a *logic* for the URM, *i.e*. a formal system whose potential may be exploited to prove properties about the semantics of URM programs and *the encoding* itself, a direction we will comment on further in the final section.

To consider the adequacy issue, we conjecture that our formalization internalizes faithfully the very initial theory developed by Cutland on paper, *i.e*. the part concerning the synthesis and the execution of individual programs. By addressing the task formally, the *soundness* of our encoding is apparent (as our programs coincide with Cutland's ones, and we have coupled to programs a formal logical system); moreover, we state a limited form of *completeness*, in the following sense.

Conjecture 5.5 *(Adequacy) Let P be an URM program and* $U=\langle i\mapsto I_i\rangle^{\iota\in[1..n]}$ *its faithful encoding. Then:*

1. *If* $P(a_1,a_2,\ldots,a_m)\downarrow b$*, then there exist* $\tau=\langle 1\mapsto b,\iota\mapsto\tau_\iota\rangle^{\iota\in[2..m]}$ *and* $\mathscr{T}((\iota\mapsto a_\iota)^{\iota\in[1..m]},U)$ *such that* $\mathscr{T}((\iota\mapsto a_\iota)^{\iota\in[1..m]},U)\Rightarrow cp_+(U,(\iota\mapsto a_\iota)^{\iota\in[1..m]},1,\tau)$
2. *If* $P(a_1,a_2,\ldots,a_m)\uparrow$*, then there exists* $\mathscr{I}((\iota\mapsto a_\iota)^{\iota\in[1..m]},U)$ *such that* $\mathscr{I}((\iota\mapsto a_\iota)^{\iota\in[1..m]},U)\Rightarrow cp_\infty(U,(\iota\mapsto a_\iota)^{\iota\in[1..m]},1)$

PROOF. *(1) By inspection on the hypothetical evaluation (to devise the termination constraint, which depends on the initial configuration* $(\iota_i\mapsto a_i)^{\iota\in[1..m]}$*), then by induction (see also Section 6). (2) By inspection on the hypothetical evaluation (to devise the invariant), then by structural coinduction.*

To conclude, we remark that, after the introduction of the very basic computability theory, Cutland develops "higher-order" methods, to devise new computable functions *without* having to write programs. It is immediate that addressing this kind of adequacy, at the moment, is out of the scope of our approach.

6 An example: partial minus

The next step of our work is to address slightly more involved concepts: in this section we exploit the formalization developed so far, by tuning it to deal with the *functions* computed by the URM.

The formal notion of *(partial) computable function* arises naturally in Cutland's presentation [12] after the preliminary definitions reported in Section 3. Namely, a program P computes a function $f\colon \mathbb{N}^m \rightharpoonup \mathbb{N}$ when, for every $a_1,a_2,\ldots,a_m,b\in\mathbb{N}^{m+1}$, the computation $P(a_1,a_2,\ldots,a_m)$ stops and b is stored in the register R_1 in the final configuration (this is written $P(a_1,a_2,\ldots,a_m)\downarrow b$) if and only if:

$$(a_1,a_2,\ldots,a_m)\in dom(f) \text{ and } f(a_1,a_2,\ldots,a_m)=b$$

A relevant application supported by our machinery is to address the *certification* of URM programs: that is, proving that a program meets the specification it is designed for. The example we will be working out in this section is the *partial* subtraction function $sub:\mathbb{N}\times\mathbb{N}\rightharpoonup\mathbb{N}$:

$$sub(m,n) \triangleq \begin{cases} m-n & \text{if } m\geq n \\ \uparrow & \text{if } m<n \end{cases}$$

An algorithm to make the URM compute this function is the following: if m and n are loaded, respectively, in R_1 and R_2, then try to let n reach m by performing *Successor* operations on R_2; correspondingly increment R_3, whose content is initially set to 0, to record the number of steps performed on R_2. This algorithm devises a *loop* in the computation, which comes to an end if and only if $m \geq n$. In any case, at any completion of the loop, the snapshot of the registers content is the following:

R_1	R_2	R_3	R_4	$\ldots$
m	$n+k$	k	0	$\ldots$

The algortithm can be implemented, for example, by the following URM program:

$$U \triangleq \langle\, 1{\mapsto}J(1,2,5),\ 2{\mapsto}S(2),\ 3{\mapsto}S(3),\ 4{\mapsto}J(1,1,1),\ 5{\mapsto}T(3,1)\,\rangle$$

The program, as required, is designed to increment in parallel r_2 and r_3 and to stop just, and only if, when $r_2{=}r_1$. It is then immediate to see that the computations under U may converge or diverge depending on the initial configuration: therefore, the implementation of the partial subtraction function has to be certified in two steps, by using the predicates cp_∞ and cp_+ defined in the previous section.

On the one hand, we prove via cp_∞ that the computation under U diverges with the configurations $(1{\mapsto}m,\ 2{\mapsto}n,\ldots)$, such that $m{<}n$ (which is the "invariant"). To complete the analysis, we establish via cp_+ that the computation under U converges to $m{-}n$ with the configurations $(1{\mapsto}m,\ 2{\mapsto}n,\ 3{\mapsto}0,\ldots)$, such that $m{\geq}n$ (this, in turn, plays the role of the "termination" constraint).

Theorem 6.1 *(Partial minus) Let $\sigma{=}(1{\mapsto}\sigma_1,\ 2{\mapsto}\sigma_2,\ 3{\mapsto}\sigma_3,\ldots)$ be a parameter configuration. Then, the implementation of the partial minus function is certified by the following properties:*

1. *(Divergence)* $\sigma_1{<}\sigma_2 \Rightarrow cp_\infty(U,\sigma,1)$
2. *(Convergence)* $\sigma_1{\geq}\sigma_2 \Rightarrow cp_+(U,\sigma,1,(1{\mapsto}\sigma_1{-}\sigma_2{+}\sigma_3,\ 2{\mapsto}\sigma_1,\ 3{\mapsto}\sigma_1{-}\sigma_2{+}\sigma_3,\ldots))$

PROOF. *(1.) By structural coinduction on the derivation. Assume the coinductive hypothesis, then evaluate the first four instructions so that the control flow loops back to the first instruction, finally apply the coinductive hypothesis and prove that the updated configuration satisfies the invariant constraint*[14]*:*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[\sigma_1{<}\sigma_2]}{\vdots}}{\sigma_1{<}\sigma_2{+}1}}{[cp_\infty(U,(1{\mapsto}\sigma_1,\ 2{\mapsto}\sigma_2{+}1,\ 3{\mapsto}\sigma_3{+}1,\ldots),1)]_{(1)}}}{cp_\infty(U,(1{\mapsto}\sigma_1,\ 2{\mapsto}\sigma_2{+}1,\ 3{\mapsto}\sigma_3{+}1,\ldots),4)}\,(jt{\cdot}r)_\infty}{cp_\infty(U,(1{\mapsto}\sigma_1,\ 2{\mapsto}\sigma_2{+}1,\ 3{\mapsto}\sigma_3,\ldots),3)}\,(s{\cdot}r)_\infty}{cp_\infty(U,(1{\mapsto}\sigma_1,\ 2{\mapsto}\sigma_2,\ 3{\mapsto}\sigma_3,\ldots),2)}\,(s{\cdot}r)_\infty}{cp_\infty(U,(1{\mapsto}\sigma_1,\ 2{\mapsto}\sigma_2,\ 3{\mapsto}\sigma_3,\ldots),1)}\,(jf{\cdot}r)_\infty}{\forall\sigma{=}(1{\mapsto}\sigma_1,\ 2{\mapsto}\sigma_2,\ 3{\mapsto}\sigma_3,\ldots).\ \sigma_1{<}\sigma_2 \Rightarrow cp_\infty(U,\sigma,1)}\,(introduction)}{\forall\sigma{=}(1{\mapsto}\sigma_1,\ 2{\mapsto}\sigma_2,\ 3{\mapsto}\sigma_3,\ldots).\ \sigma_1{<}\sigma_2 \Rightarrow cp_\infty(U,\sigma,1)}\,(1)$$

[14] See Section 2 about the conventions for displaying CC$^{(Co)Ind}$ top-down proofs in natural deduction style.

(2.) By induction on $p{=}\sigma_1{-}\sigma_2$. If $p{=}0$, the evaluation of the program U reduces to obeying just the first instruction (the Jump condition is true) and the last one, hence the thesis is immediate. If $p{=}q{+}1$, the evaluation of the first four instructions causes the control flow to loop back to the first instruction, with the configuration $(1{\mapsto}\sigma_1,\ 2{\mapsto}\sigma_2{+}1,\ 3{\mapsto}\sigma_3{+}1,\ldots)$; the thesis follows from the inductive hypothesis.

Finally, choosing $\sigma_3{=}0$ implies the convergence of the computation under U with σ to $\sigma_1{-}\sigma_2$.

Inductive versus coinductive evaluations. Regarding *partial* functions, it is apparent that the two predicates cp_+ and cp_∞ act as complementary, being the first one responsible for the treatment of the elements in the domain of the function involved and the second one for all the extra computations.

About this separation between inductive and *purely* coinductive evaluations, we wish to remark that it has not been possible to deal with the semantics of URM programs by using a unique, *potentially* coinductive judgment. Actually, by restricting *e.g.* on abstract programs, if such a predicate was defined through the rules $(f{\cdot}l)_+$, $(t{\cdot}l)_+$, $(f{\cdot}r)_\infty$ and $(t{\cdot}r)_\infty$ of Definition 4.1, would be too weak. Far from being an obstacle for our goals, this fact has caused just to double a part of the encoding, to define both cp_+ and cp_∞; in any case, such a solution provides with an extra proof principle, *i.e.* the possibility of carrying out proofs by structural induction on the derivation of converging computations.

Nevertheless, these considerations about the relationship between inductive, potential and pure coinductive evaluation point out the need of further research efforts, along the lines pursued by the much more advanced work by Leroy and Grall [22] (see the next section for the discussion of related work).

7 Further and related work

In this document we have given an account of an experiment in $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$, about modeling and reasoning on the execution of converging and diverging low-level, assembly-like programs, carried out by the Unlimited Register Machine (URM) [12]. The particular perspective which has inspired our research is the formalization of a workbench to certify the implementation of the functions computed by the URM; as a proof of concept, we have addressed the partial minus function on natural numbers. The encoding technique needed to accomplish our goal is quite plain, apart from the use of the coinduction: in fact, we have taken most advantage of the (co)inductive specification and proof principles provided by the $\mathrm{CC}^{(\mathrm{Co})\mathrm{Ind}}$ intuitionistic type theory and mechanized in the Coq proof assistant [17, 26].

In this final section we sketch some hints to exploit the potential of our formalization, along two main directions: computability and traces of execution.

Computability. In our work we have mastered the very basic computability theory of the URM: essentially, we are able to prove that *specific* URM programs implement the functions they are designed for. So we have coupled a *logic*, whose mechanization is supported by Coq, to the bare URM. Nevertheless, exploiting the machinery requires a non-trivial analysis and practice by the user, who has to pick out ad-hoc properties (*termination* and *invariant* conditions) to achieve the certification of URM code.

At this point, to pursue at a deeper extent the formalization of the computability theory, one has to change a bit perspective, gaining a more abstract level. This opens actually two new directions, which form the core of the computability: lifting from programs to functions (which they implement) and describing "higher-order" methods, to combine such functions for obtaining new, more sophisticated computable functions. Therefore, one should add at least a new meta-level, where partial functions are first-class citizens. A possible approach towards this goal is to investigate more abstract properties of URM programs, such as *equivalence*. This effort, in turn, would open further research lines, and tends

again, as invariance does, to the objective of capturing not only the outcome of the execution of programs, but also the observable effects.

As far as we know, there is no related work about formalizing the historical models used to develop the computability theory (and the URM, in particular). We see this as a serious gap from the point of view of certified mathematics, a framework where the research is nowadays intense; hence the present document is also an effort to contribute closing this gap.

Traces of execution. Leroy and Grall [22] adopt coinduction within $CC^{(Co)Ind}$ to capture both finite and infinite evaluations of a *call-by-value λ-calculus*. The motivation of that work is the attempt to describe big-step semantics by coinduction, because big-step semantics is more convenient than small-step to prove the correctness of program transformations, such as *compilation*. Nevertheless, big-step semantics is traditionally defined by induction, thus allowing to describe only terminating evaluation.

Grall and Leroy prove that (only) a big-step semantics that separates terminating evaluation (described by an inductive predicate) from diverging evaluation (described by a purely coinductive predicate) corresponds exactly to finite and non-finite small-step reductions. Afterwards, the authors extend both the semantics to produce not only the outcome of an evaluation (convergence and output, or divergence) but also an *execution trace*, in the form of a potentially infinite sequence of terms representing the intermediate reducts of the source program. This extension is fundamental to establish semantic preservation properties for program transformation (such as compilation) and is very important to investigate observational equivalence for imperative languages.

Therefore, it would be stimulating to experiment with traces of execution for the URM (for example in the form of potential infinite sequences of configurations) to address *e.g.* equivalence of programs.

Other work related to divergence or low-level languages. There are several contributions in the literature exploiting the potential of coinductive definitions and proofs within $CC^{(Co)Ind}$ to master the fundamental concept of non-terminating computation. Some of these approaches concern transition systems [10, 3], linear temporal logic [9, 3] and process algebras [18, 20].

Finally, from a complementary point of view, we observe that in recent years the metatheory of low-level machines has been studied by several authors in more realistic settings [11, 25, 6].

References

[1] H. Barendregt & T. Nipkow, editors (1994): *Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24-28, 1993, Selected Papers. Lecture Notes in Computer Science* 806, Springer.

[2] S. Berardi & M. Coppo, editors (1996): *Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, June 5-8, 1995, Selected Papers. Lecture Notes in Computer Science* 1158, Springer.

[3] Y. Bertot & P. Castéran (2004): *Interactive Theorem Proving and Program Development, Coq'Art:the Calculus of Inductive Constructions*. Springer-Verlag.

[4] Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin & L. Théry, editors (1999): *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings. Lecture Notes in Computer Science* 1690, Springer.

[5] Y. Bertot & E. Komendantskaya (2009): *Using Structural Recursion for Corecursion*. In S. Berardi, F. Damiani & U de'Liguoro, editors: *Types for proofs and programs 2008, Lecture Notes in Computer Science* 5497, Springer, pp. 220–236. Available at `http://hal.inria.fr/inria-00322331`.

[6] A. Chlipala (2007): *A certified type-preserving compiler from lambda calculus to assembly language*. In Ferrante & McKinley [14], pp. 54–65. Available at http://doi.acm.org/10.1145/1250734.1250742.

[7] A. Ciaffaglione (2011): *The Web Appendix of this paper*. Available at http://www.dimi.uniud.it/ciaffagl.

[8] T. Coquand (1993): *Infinite Objects in Type Theory*. In Barendregt & Nipkow [1], pp. 62–78. Available at http://dx.doi.org/10.1007/3-540-58085-9_72.

[9] S. Coupet-Grimal (2003): *An Axiomatization of Linear Temporal Logic in the Calculus of Inductive Constructions*. *J. Log. Comput.* 13(6), pp. 801–813. Available at http://dx.doi.org/10.1093/logcom/13.6.801.

[10] S. Coupet-Grimal & L. Jakubiec (1999): *Hardware Verification Using Co-induction in COQ*. In Bertot et al. [4], pp. 91–108. Available at http://dx.doi.org/10.1007/3-540-48256-3_7.

[11] K. Crary (2003): *Toward a foundational typed assembly language*. In: *POPL*, pp. 198–212. Available at http://doi.acm.org/10.1145/640128.604149.

[12] N. J. Cutland (1980): *Computability: An Introduction to Recursive Function Theory*. Cambridge University Press.

[13] P. Dybjer, B. Nordström & J. M. Smith, editors (1995): *Types for Proofs and Programs, International Workshop TYPES'94, Båstad, Sweden, June 6-10, 1994, Selected Papers*. *Lecture Notes in Computer Science* 996, Springer.

[14] J. Ferrante & K. S. McKinley, editors (2007): *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. ACM.

[15] H. Geuvers & F. Wiedijk, editors (2003): *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002, Selected Papers*. *Lecture Notes in Computer Science* 2646, Springer.

[16] P. Di Gianantonio & M. Miculan (2002): *A Unifying Approach to Recursive and Co-recursive Definitions*. In Geuvers & Wiedijk [15], pp. 148–161. Available at http://dx.doi.org/10.1007/3-540-39185-1_9.

[17] E. Giménez (1994): *Codifying Guarded Definitions with Recursive Schemes*. In Dybjer et al. [13], pp. 39–59. Available at http://dx.doi.org/10.1007/3-540-60579-7_3.

[18] E. Giménez (1995): *An Application of Co-inductive Types in Coq: Verification of the Alternating Bit Protocol*. In Berardi & Coppo [2], pp. 135–152. Available at http://dx.doi.org/10.1007/3-540-61780-9_67.

[19] E. Giménez (1998): *Structural Recursive Definitions in Type Theory*. In Larsen et al. [21], pp. 397–408. Available at http://dx.doi.org/10.1007/BFb0055070.

[20] F. Honsell, M. Miculan & I. Scagnetto (2001): *pi-calculus in (Co)inductive-type theory*. *Theor. Comput. Sci.* 253(2), pp. 239–285. Available at http://dx.doi.org/10.1016/S0304-3975(00)00095-5.

[21] K. Guldstrand Larsen, S. Skyum & G. Winskel, editors (1998): *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings*. *Lecture Notes in Computer Science* 1443, Springer.

[22] X. Leroy & Hervé Grall (2009): *Coinductive big-step operational semantics*. *Inf. Comput.* 207(2), pp. 284–304. Available at http://dx.doi.org/10.1016/j.ic.2007.12.004.

[23] J. C. Shepherdson & H. E. Sturgis (1963): *Computability of Recursive Functions*. *J. ACM* 10(2), pp. 217–255. Available at http://doi.acm.org/10.1145/321160.321170.

[24] B. Steffen & G. Levi, editors (2004): *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, January 11-13, 2004, Proceedings*. *Lecture Notes in Computer Science* 2937, Springer.

[25] G. Tan, A. W. Appel, K. N. Swadi & D. Wu (2004): *Construction of a Semantic Model for a Typed Assembly Language*. In Steffen & Levi [24], pp. 30–43. Available at http://dx.doi.org/10.1007/978-3-540-24622-0_4.

[26] The Coq Development Team (2010): *The Coq Proof Assitant Reference Manual, version 8.3*. INRIA.